1975 ACM Student Award
Paper: Second Place

Multidimensional Binary Search Trees Used for Associative Searching

Jon Louis Bentley
Stanford University

This paper develops the multidimensional binary search tree (or k-d tree, where k is the dimensionality of the search space) as a data structure for storage of information to be retrieved by associative searches. The k-d tree is defined and examples are given. It is shown to be quite efficient in its storage requirements. A significant advantage of this structure is that a single data structure can handle many types of queries very efficiently. Various utility algorithms are developed; their proven average running times in an n record file are: insertion, $O(\log n)$; deletion of the root, $O(n^{(k-1)/k})$; deletion of a random node, $O(\log n)$; and optimization (guarantees logarithmic performance of searches), $O(n \log n)$. Search algorithms are given for partial match queries with t keys specified [proven maximum running time of $O(n^{(k-t)/k})$] and for nearest neighbor queries [empirically observed average running time of $O(\log n)$.] These performances far surpass the best currently known algorithms for these tasks. An algorithm is presented to handle any general intersection query. The main focus of this paper is theoretical. It is felt, however, that k-d trees could be quite useful in many applications, and examples of potential uses are given.

Key Words and Phrases: associative retrieval, binary search trees, key, attribute, information retrieval system, nearest neighbor queries, partial match queries, intersection queries, binary tree insertion

CR Categories: 3.63, 3.70, 3.74, 4.49

This paper was awarded Second Place in ACM's 1975 George E. Forsythe Student Paper Competition.

Author's present address: Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27514.

1. Introduction

The problem of associative retrieval (often referred to as *retrieval by secondary key*) centers around a file F which is a collection of records. Each record R of F is an ordered k-tuple $(v_0, v_1, \ldots, v_{k-1})$ of values which are the *keys*, or *attributes*, of the record. A retrieval of records from F is initiated at the request of the user, which could be either mechanical or human. A retrieval request is called a *query* of the file, and specifies certain conditions to be satisfied by the keys of the records it requests to be retrieved from F. An information retrieval system must be capable of initiating an appropriate search upon the arrival of a query to that system. If a query is allowed to specify conditions dealing with a multiplicity of the keys, the searches performed by the system are considered associative. If the user of the system is restricted to specifying conditions for only one of the keys, the resulting search is not considered to be associative (in that case only one of the keys is considered to be "the key" and the remaining attributes are referred to as "data").

Numerous methods exist for building an information retrieval system capable of handling such associative queries. Among these are inverted files, methods of compounding attributes, superimposed coding systems, and combinatorial hashing. Knuth [5] discusses these ideas in detail. McCreight [6] has proposed that the keys be "shuffled" together bitwise; then unidimensional retrieval algorithms could be used to answer certain queries. Rivest investigated in his thesis [7] the use of binary search tries (see [5] for a detailed discussion of tries) to store records when they are composed of binary keys. Finkel and Bentley [3] discuss a data structure, quad trees, that stores the records of the file in a tree whose nodes have out-degree of 2^k; theirs was the first general approach to use a tree structure. None of the above approaches seem to provide a "perfect" environment for associative retrieval. Each of them falls short in some very important way, either in having only a small class of queries easily performed, large running time, large space requirements, horrible worst cases, or some other adverse properties.

This paper presents a new type of data structure for associative searching, called the multidimensional binary search tree or k-d tree, which is defined in Section 2. In Section 3 an efficient insertion algorithm is given and analyzed. Many types of associative searches are discussed in Section 4, and the k-d tree is shown to perform quite well in all of them. Its worst case performance in partial match searches is analyzed and is shown to equal the best previously attained average performance. A search algorithm is presented that can answer any intersection query. An algorithm given seems to solve the nearest neighbor problem in logarithmic time; its running time is much less than any other known method. In Section 5 deletion is shown to be possible in k-d trees, and it is analyzed. Section 6 dis-

cusses an optimization algorithm that efficiently transforms any collection of records into a k-d tree with optimal properties. With this background, a discussion of potential applications of k-d trees is presented in Section 7. Areas for further work are discussed in Section 8, and conclusions are drawn in Section 9.

2. Definitions and Notations

If a file is represented as a k-d tree, then each record in the file is stored as a node in the tree. In addition to the k keys which comprise the record, each node contains two pointers, which are either null or point to another node in the k-d tree. (Note that each pointer can be considered as specifying a subtree.) Associated with each node, though not necessarily stored as a field, is a discriminator, which is an integer between 0 and $k - 1$, inclusive. Let us define a notation for these items: the k keys of node P will be called $K_0(P), \ldots, K_{k-1}(P)$, the pointers will be $LOSON(P)$ and $HISON(P)$, and the discriminator will be $DISC(P)$. The defining order imposed by a k-d tree is this: For any node P in a k-d tree, let j be $DISC(P)$, then for any node Q in $LOSON(P)$, it is true that $K_j(Q) < K_j(P)$; likewise, for any node R in $HISON(P)$, it is true that $K_j(R) > K_j(P)$. (This statement does not take into account the possibility of equality of keys, which will be discussed shortly.)

All nodes on any given level of the tree have the same discriminator. The root node has discriminator 0, its two sons have discriminator 1, and so on to the kth level on which the discriminator is $k - 1$; the $(k + 1)$-th level has discriminator 0, and the cycle repeats. In general, $NEXTDISC$ is a function defined as

$$NEXTDISC(i) = (i + 1) \bmod k,$$

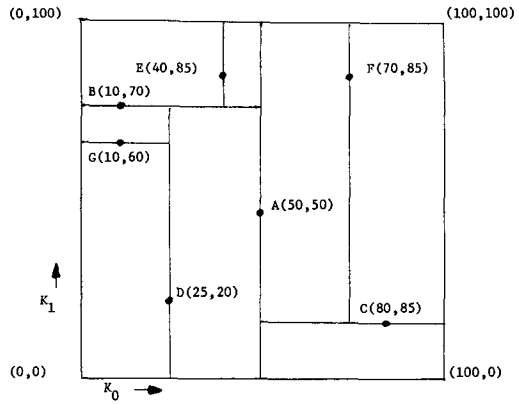
and $NEXTDISC(DISC(P)) = DISC(LOSON(P))$, and likewise for $HISON(P)$ (if they are non-null). Figure 1 gives an example of records in 2-space stored as nodes in a 2-d tree.

The problem of equality of keys mentioned above arises in the definition of a function telling which son of P's to visit next while searching for node Q. This function is written $SUCCESSOR(P, Q)$ and returns either $LOSON$ or $HISON$. Let j be $DISC(P)$; if $K_j(P) \neq K_j(Q)$, then it is clear by the defining property of k-d trees which son $SUCCESSOR$ should return. If the two K_j's are equal, the decision must be based on the remaining keys. The choice of decision function is arbitrary, but for most applications the following method works nicely: Define a superkey of P by

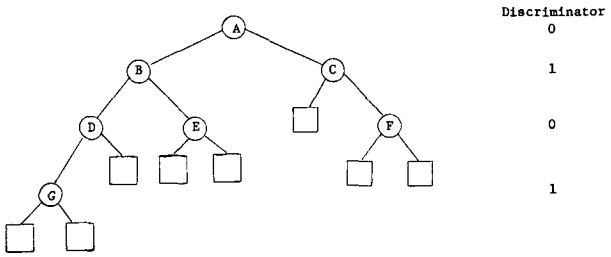
$$S_j(P) = K_j(P)K_{j+1}(P) \ldots K_{k-1}(P)K_0(P) \ldots K_{j-1}(P),$$

the cyclical concatenation of all keys starting with K_j. If $S_j(Q) < S_j(P)$, $SUCCESSOR$ returns $LOSON$; if $S_j(Q) > S_j(P)$, it returns $HISON$. If $S_j(Q) = S_j(P)$ then all k keys are equal, and $SUCCESSOR$ returns a special value to indicate that.

Fig. 1. Records in 2-space stored as nodes in a 2-d tree.
Records in 2-space stored as nodes in a 2-d tree (boxes represent range of subtree):



Planar graph representation of the same 2-d tree (*LOSON*'s are expressed by left branches, *HISON*'s by right branches, and null sons by boxes):



Let us define node Q to be j-greater than node P if and only if $SUCCESSOR(P, Q)$ is $HISON$, and to be j-less than node R if and only if $SUCCESSOR(R, Q)$ is $LOSON$. It is obvious how these definitions can be used to define the j-maximum and j-minimum elements of a collection of nodes. The j-distance between nodes P and Q is $| K_j(P) - K_j(Q) |$, and the j-nearest node of a set S to node P is the node Q in S with minimum j-distance between P and Q. If a multiplicity of the nodes have minimum j-distance from P, then the j-nearest node is defined as the j-minimum element in the subset of closest nodes which are j-greater than P (or similarly, the j-maximum element of the nodes which are j-less than P).

We will typically use n to be the number of records in the file which is stored as a k-d tree, and therefore the number of nodes in the tree. We have already used k as the dimensionality of the records.

The keys of all nodes in the subtree of any node, say P, in a k-d tree are known by P's position in the tree to be bounded by certain values. For instance, if P is in the $HISON$ subtree of Q, and $DISC(Q)$ is j, then all nodes in P are j-greater than Q; so for any R in $HISON(P)$, $K_j(R) \geq K_j(P)$. To employ this knowledge in our algorithms, we will define a *bounds array* to hold the information. If B is a bounds array associated with node P, then B has $2k$ entries, $B(0), \ldots, B(2k - 1)$. If Q is a

descendant of P, then it is true for all integers $j \in [0, k-1]$ that $B(2j) \leq K_j(Q) \leq B(2j+1)$. Bounds array B is considered to be initialized if for all integers $j \in [0, k-1]$, $B(2j) = -\infty$ and $B(2j+1) = \infty$. For example, the bounds array representing node C in Figure 1 is (50, 100, 0, 100). This indicates that all k_0 values in the subtree are bounded by 50 and 100, and all k_1 values are bounded by 0 and 100.

It was noted that it is not necessary to store the discriminator as a field in each node, and one can see that it is easy to keep track of what kind of discriminator one is visiting as one descends in a k-d tree. With the idea in mind that it is superfluous to do so, we will store the discriminator in each node to make the algorithms we write more easily understandable.

3. Insertion

In this section we will first describe an algorithm that inserts a node into a k-d tree. We will then analyze k-d trees and show that if the algorithm is used to insert random nodes into an initially empty tree the resulting tree will have the nice properties of a randomly built one-dimensional binary search tree.

3.1 An Insertion Algorithm

The algorithm used to insert a node into a k-d tree is also used to search for a specific record in a tree. It is passed by a node, say P. If P is in the tree, the algorithm returns a pointer to P, and if P is not in the tree it returns Λ and inserts P into the tree. Algorithm *INSERT* describes one way of performing such an operation.

Algorithm *INSERT* (k-d tree search and insertion)

This algorithm is passed a node P, which is not in the tree (its *HISON*, *LOSON*, and *DISC* fields are not set). If there is a node in the tree with equal keys, the address of that node is returned; otherwise the node is inserted into the tree and Λ is returned.

I1. [Check for null tree.] If $ROOT = \Lambda$ then set $ROOT \leftarrow P$, $HISON(P) \leftarrow \Lambda$, $LOSON(P) \leftarrow \Lambda$, $DISC(P) \leftarrow 0$, and return Λ; otherwise, set $Q \leftarrow ROOT$ (Q will move down the tree).

I2. [Compare.] If $K_i(P) = K_i(Q)$ for $0 \leq i \leq k-1$ (i.e. the nodes are equal) then return Q. Otherwise, set $SON \leftarrow SUCCESSOR(Q,P)$ (*SON* will be *HISON* or *LOSON*). If $SON(Q) = \Lambda$, then go to I4.

I3. [Move down.] Set $Q \leftarrow SON(Q)$ and go to I2.

I4. [Insert new node in tree.] Set $SON(Q) \leftarrow P$, $HISON(P) \leftarrow \Lambda$, $LOSON(P) \leftarrow \Lambda$, $DISC(P) \leftarrow NEXTDISC(DISC(Q))$, return Λ.

3.2 Analysis of Randomly Built k-d Trees

Consider a given binary tree of n nodes; our goal in this analysis of k-d trees is to show that the probability of constructing that tree by inserting n random nodes into an initially empty k-d tree is the same as the probability of attaining that tree by random insertion into a one-dimensional binary search tree. Once we have shown this to be true, the theorems which have been proved about one-dimensional binary search trees will be applicable to k-d trees.

We must first define what we mean by random nodes. Since only the relative magnitudes of the keys and the order in which the records arrive are relevant for purposes of insertion, we can assume that the records to be inserted will be defined by a k-tuple of permutations of integers $1, \ldots, n$. Then the first record to be inserted, say P, would be defined by $K_0(P)$, the first element in the first permutation, and so on, to $k_{k-1}(P)$, the first element in the kth permutation. The nodes will be considered random if all of the $(n!)^k$ k-tuples of permutations are equally likely to occur.

Let us give each of the n nodes in the binary tree t a unique identification number which is an integer between 1 and n. Define S_i as the number of nodes in the subtree of t whose root is node i. To simplify our discussion of null sons let us define the identification number of a null node to be $n+1$; thus $S_{n+1} = 0$. We will use L_i as the number of nodes in the left subtree (or *LOSON*) of node i, and H_i as the number of nodes in the right subtree (or *HISON*); note $S_i = L_i + R_i + 1$.

It is important to observe the following fact about the ordering in a collection of random nodes that are to be made into a k-d tree. The first node in the collection, say P (which is the first to be inserted in the tree), will become the root. This induces a partition of the remaining nodes into two subcollections: those nodes that will be in P's left (*LOSON*) subtree, and those that will be in the right (*HISON*) subtree. If Q falls in the right subtree of P, and R falls in P's left subtree, then their relevant ordering (that is, whether or not Q precedes R) in the original collection is unimportant. The same tree would be built if P was the first element in the collection, and then came all the nodes that fell in P's right subtree, followed by all the nodes that fell in P's left subtree, as long as the orderings in the left and right subcollections were maintained. A second important fact we will use is that when a collection of random nodes is split into two subcollections by this process, the resulting subcollections are themselves random collections of nodes. This is due to the original independence of keys in a given record; the partitioning in no way disturbs the independence.

After having made these two observations, it is easy to compute the probability that tree t as described above results when n random nodes are inserted into an initially empty k-d tree. Assume that the root is a j-decider and that its identification number is i; then $S_i = n$. The probability that the first record will partition the collection into two subcollections having respective sizes L_i and R_i is the probability that the jth key of the first element is the (L_i+1)-th in the set of all jth keys. Because of the random nature of the nodes (all of the nodes are equally likely to be the first in the collection), this probability is $1/S_i$. Now we have reduced the problem; we know that the probability of t occurring is $1/S_i$ times the probability of both the subcollections

forming their respective subtrees. By the second observation above, these probabilities are independent of the choice of the root of the tree and of one another. Therefore we can split the nodes into the left and right subcollections (by the first observation) and apply this analysis recursively to the left son and right son, doing so as we visit each node in the tree once and only once. It is clear that the probability of t resulting is

$$P_k(t) = \prod_{1 \le i \le n} 1/S_i .$$

We know that the standard binary search tree is a 1-d tree. Since $P_k(t)$ is independent of k, the probability of attaining t by inserting random nodes in a k-d tree must be the same as that of attaining t by inserting random nodes in a standard binary search tree. Indeed, the above formula for $P_k(t)$ was given by Knuth [5] as the probability of attaining t by random insertion in a standard binary search tree. We can now apply two of the results in [5] to k-d trees. Let C_n be the number of nodes visited to find a node in a k-d tree of size n. Then we know that the mean of the distribution of C_n is

$$Mean(C_n) = 2(1 + 1/n)H_n - 3 \simeq 1.386 \log_2 n$$

and the variance is

$$Var(C_n) = 7n^2 - 4(n + 1)^2 H_n^{(2)} - 2(n + 1)H_n + 13n.$$

Here we have used the functions

$$H_n^{(x)} = \sum_{1 \le k \le n} 1/k^x$$

and

$$H_n = H_n^{(1)}.$$

Thus we know that typical insertions and record look-ups in a k-d tree will examine approximately $1.386 \log_2 n$ nodes.

4. Searching

Searches are typically initiated in response to a query expressed in relation to the set of all valid records. The purpose of a search is to find in the data structure the records specified by the query. It is therefore reasonable to classify searches by the types of queries which invoke them. We will follow in the spirit of Rivest [7] as we make the primary distinction between intersection queries and best-match queries in our investigation of searching in k-d trees.

4.1 Intersection Queries

Intersection queries are so named because they specify that the records to be retrieved are those that intersect some subset of the set of valid records. This is by far the most common type of query. The specifications of the sets in which are all records to be retrieved can range from simply defined sets such as $\{P \mid K_3(P) = 7\}$ to complexly defined sets like $\{P \mid [(1 \le K_1(P) \le 5) \wedge (2 \le K_2(P) \le 4)] \vee (K_7(P) = 8)\}$. We will examine search strategies for three increasingly complex query types, each embodying its predecessors as special cases. The region search, the third type we will examine, is capable of searching for all records specifiable by any intersection query.

4.1.1 Exact Match Queries

The simplest type of query is the exact match query (called a "simple query" by Knuth [5], and a "point search" by Finkel and Bentley [3]), which asks if a specific record is in the data structure. The search algorithm to determine this (Algorithm *INSERT*) and its analysis are presented in Section 3. The only thing that remains to be said regarding the topic of exact match searching is this: If the exact match query is the only type of query to be posed, k-d trees should not be used as the data structure to store the records. Though to the user the keys appear to be independent, they should be merged together into one superkey and a more well known data structure for unidimensional storage and retrieval should be employed.

4.1.2 Partial Match Queries

The next more complex type of intersection query is one in which values are specified for a proper subset of the keys. If values are specified for t keys, $t < k$, then the query is called a "partial match query with t keys specified." Assume that $\{s_i\}$ and $\{v_i\}$ are sets such that the keys specified are $K_{s_1}, K_{s2}, \ldots, K_{st}$, and the values they must have to be a valid response to the query are $v_{s_1}, v_{s_2}, \ldots, v_{s_t}$. The set of points for which we are searching is then $\{P \mid K_{s_i}(P) = v_{s_i}$ for $1 \le i \le t\}$.

Rivest has studied this problem quite thoroughly for the case of binarily valued keys. He proposed that binary search tries be used to store the data. His "standard compact" tries are roughly identical to "bit-key" k-d trees. The reader interested in the binary case is referred to Rivest's thesis [7]; it contains a description of the behavior of tries (and, equivalently, k-d trees) for this application. The results presented here parallel those in that work.

A recursive search algorithm to find all nodes satisfying a partial match query for continuously valued keys is easy to define (it is shown in Section 4.1.3 that the *REGIONSEARCH* algorithm presented here is capable of efficiently performing a partial match search, so we will merely sketch an algorithm here.) On each level of recursion the algorithm is passed a node, say P. If P satisfies the query it is reported. Let us suppose that P is a J-discriminator; there are now two cases we must handle. If $J = s_i$ for some i, then we need only continue our search down one subtree of P: if $v_{s_i} < K_J(P)$ we go down $LOSON(P)$, if $v_{s_i} > K_J(P)$ we go down $HISON(P)$, and if the values are equal we go down one of the subtrees depending on the definition of successor. If $J \notin \{s_i\}$ then we must continue the search down both

of P's subtrees. (Implicit in the algorithm is the fact that the search continues down a subtree only if that subtree is non-null, i.e. P's *SON* field is not Λ.)

Let us now analyze the number of nodes visited by the algorithm in doing a partial match search with t keys specified in an "ideal" k-d tree of n nodes; a tree in which $n = 2^{kh} - 1$ and all leaves appear on level kh. (It is shown in Section 6 that such a tree is attainable for any collection of $2^{kh} - 1$ nodes by using algorithm *OPTIMIZE*.) The search algorithm starts by visiting one node, the root, and the number of nodes visited on the next level grows by a factor of 1 (if $J = s_i$, we go down only one subtree of each node, hence visiting the same number on the next level) or two (if $J \notin \{s_i\}$, we have to visit both subtrees of the node, thereby visiting twice as many nodes on the next level). Since the growth rate at any level is a function of the discriminator of that level, and the discriminators are cyclic with cycle k, the growth rate will be cyclic as well. The pessimal arrangement of keys is to have these unspecified as the first $k - t$ keys in the cycle, postponing any pruning until after a small geometric explosion. The maximum number of nodes visited during the whole search will therefore be the sum of the following series:

$$\left.\begin{array}{l} \overbrace{\underbrace{1 + 2 + \cdots + 2^{k-t-1}}_{k-t \text{ elements}} + \underbrace{2^{k-t-1} + \cdots + 2^{k-t-1}}_{t \text{ elements}}}^{k \text{ elements}} \\ + 2^{k-t} + \cdots + 2^{2(k-t)-1} + 2^{2(k-t)-1} + \cdots + 2^{2(k-t)-1} + \cdots \\ \quad \vdots \\ \quad + 2^{(h-1)(k-t)} + \cdots + 2^{h(k-t)-1} \\ \quad + 2^{h(k-t)-1} + \cdots + 2^{h(k-t)-1}. \end{array}\right\} h \text{ levels}$$

We can now sum this series to calculate $V(n, t)$, the maximum number of nodes visited during a partial match search with t keys specified in an ideal tree of n nodes. (For brevity, we will here define $m = k-t$.)

$$\begin{aligned} V(n, t) &= \sum_{0 \le i \le h-1} 2^{mi} \left[\left(\sum_{0 \le j \le m-1} 2^j\right) + 2^{m-1}t\right] \\ &= \sum_{0 \le i \le h-1} 2^{mi}[2^m - 1 + 2^{m-1}t] \\ &= [(t+2)2^{m-1} - 1] \sum_{0 \le i \le h-1} 2^{mi} \\ &= [(t+2)2^{m-1} - 1] \frac{2^{mh} - 1}{2^m - 1} \\ &= \frac{[(t+2)2^{m-1} - 1]}{2^m - 1} (2^{mh} - 1). \end{aligned}$$

Since we know $2^{mh} = 2^{(m/k)kh} = (2^{kh})^{m/k} = (n+1)^{m/k}$, we see

$$V(n, t) = \frac{[(t+2)2^{m-1} - 1]}{2^m - 1} [(n+1)^{m/k} - 1].$$

The amount of work done in any partial match search with t keys specified in an ideal tree of n nodes is therefore $cn^{m/k} + d$ for some small constants c and d. This has been conjectured by Rivest [7] to be a lower bound for the average amount of work done in a partial match search; by construction we have shown this to be an upper bound not only for the average but for all partial match queries.

All of our analysis has been for the case of the perfectly balanced tree; the one in which we might expect to have the fastest searches. However, Rivest [7] has shown that the perfectly balanced trees have the *highest* average retrieval time. Therefore the results that we have shown are an expected upper bound on the retrieval time required by the algorithm.

4.1.3 Region Queries

The most general type of intersection query is one in which any region at all may be specified as the set with which the records to be retrieved must intersect, hence its name "region query." This query is the same as the "region search" query described by Finkel and Bentley [3] and facilitates the range, best match with restricted distance, and boolean queries described by Knuth [5] and Rivest [7]. Any subset of the set of valid records region can be specified in a region query, so it is therefore the most general intersection query possible. (An exact match query corresponds to the region being a point, a partial match query with t keys specified corresponds to the region being a $k - t$ dimensional hyperplane.)

The algorithm to accomplish a region search need not specifically know the definition of the region in which it is searching; rather it finds out all it needs to know by calling two functions which describe the region. The first, *IN_REGION*, is passed a node in the tree and returns true if and only if that node is contained in the region. The function *BOUNDS_INTERSECT_REGION* is passed a bounds array and returns true if and only if the region intersects the hyper-rectangle described by the bounds array. Nor does the algorithm know what to do once it finds a node in the region; it calls procedure *FOUND* to report all nodes it finds in the region. A recursive definition of the general intersection query search algorithm is given below. It would be invoked initially by the command *REGIONSEARCH(ROOT, B)*, where B is a bounds array initialized as described in Section 2. [See next page.]

Algorithm *REGIONSEARCH* uses the bounds stored at each node of the tree to determine whether it is possible that any descendants of the node might lie in the region being searched. A subtree is visited by the algorithm if and only if this possibility exists. Consequently, the algorithm visits as few nodes as possible, given the limited information stored at each node. In this sense, *REGIONSEARCH* is an optimal algorithm for region searches in k-d trees as we have described them.

The versatility of algorithm *REGIONSEARCH* makes its formal analysis extremely difficult. Its per-

(*cont'd in col. 2, next page*)

Algorithm *REGIONSEARCH* (Search a k-d tree for all points contained in a specified region.)

This recursive algorithm is passed a node P and a bounds array B that specifies the bounds for P's descendants. It assumes the existence of the three procedures *IN_REGION*, *FOUND*, and *BOUNDS_INTERSECT_REGION*. It reports all nodes in the subtree whose root is P which are in the region by invoking the *FOUND* procedure.

R1. [Is P in the region?] If *IN_REGION*(P) then call *FOUND*(P) (if the node is in the region, then report that fact by calling *FOUND* with parameter P).

R2. [Improve the bounds for the subtrees.] Allocate B_L, B_H as bounds arrays, and copy the array B into both. Set $J \leftarrow DISC(P)$ (J is the dimension of the bounds to be changed.) Set $B_L(2J+1) \leftarrow K_J(P)$, $B_H(2J) \leftarrow K_J(P)$. (This step notes that $K_J(P)$ is a J-upper bound of the nodes in the *LOSON* subtree and a J-lower bound of the nodes in the *HISON* subtree.)

R3. [Search *LOSON* subtree.] If $LOSON(P) \neq \Lambda$ and *BOUNDS_INTERSECT_REGION*(B_L) then *REGIONSEARCH*(*LOSON*(P), B_L).

R4. [Search *HISON* subtree.] If $HISON(P) \neq \Lambda$ and *BOUNDS_INTERSECT_REGION*(B_H) then *REGIONSEARCH*(*HISON*(P), B_H).

As an example of *IN_REGION* and *BOUNDS_INTERSECT_REGION* functions, the following pseudo-Algol procedures are defined for a hyper-rectangular region.

Pseudo-Algol *IN_REGION* and *BOUNDS_INTERSECT_REGION* procedures for a rectilinearly oriented hyper-rectangular region defined by a bounds array *RECDEF*.

```
boolean procedure IN_REGION (node P):
begin
comment returns true iff P is in the hyper-rectangle defined by
  RECDEF;
for I ← 0 step 1 until k − 1 do
  begin
  if K_I(P) < RECDEF(2·I) then
    return false;
  if K_I(P) > RECDEF(2·I+1) then
    return false
  end;
return true
end;

boolean procedure BOUNDS_INTERSECT_REGION (array B);
begin
comment returns true iff the hyper-rectangle defined by bounds
  array B intersects the hyper-rectangle defined by RECDEF;
for I ← 0 step 2 until 2·(k−1) do
  begin
  ifB(I) > RECDEF(I+1) then
    return false;
  if B(I+1) < RECDEF(I) then
    return false
  end;
return true
end;
```

Similar procedures can be written for many other k-dimensional geometric regions. The logical functions AND, OR, and NOT can then be used to implement searches in intersections, unions, and complements of basic regions.

(*cont'd from page 513*)

formance in any given situation will certainly depend on type and size of the region which it is searching. Limited empirical tests show that the algorithm performs reasonably well in searching hyper-rectangular regions. Relevant empirical data appear in the discussion of region searching in quad trees by Finkel and Bentley [3]. The similarity between k-d trees and quad trees and between the corresponding *REGIONSEARCH* algorithms make Table 3 in that paper quite useful in estimating the amount of work the *REGIONSEARCH* algorithm does in searching hyper-rectangular regions.

4.2 Nearest Neighbor Queries

Given a distance function D, a collection of points B (in k-dimensional space), and a point P (in that space), it is often desired to find P's nearest neighbor in B. The nearest neighbor is Q such that

$$(\forall R \leq B)\ \{(R \neq Q) \Rightarrow [D(R, P) \geq D(Q, P)]\}.$$

A similar query might ask for the m nearest neighbors to P.

[Ed. Note. In the original versions of this paper, an algorithm to answer such a query was presented. Empirical tests showed that its running time is logarithmic in n. The algorithm was quite difficult to understand and efficient only for the Minkowski ∞ metric (the maximum coordinate metric). A recent paper by Friedman, Bentley, and Finkel [1] gives a more easily understood version of the algorithm which uses a slightly modified form of k-d trees. The modified algorithm is defined recursively and is efficient for any Minkowski p metric. Analysis shows that the number of nodes visited by the algorithm is proportional to $\log_2 n$ and the number of distance calculations to be approximately $m2^k$. For economies of space, we have deleted the section of this paper on nearest neighbor searching. Interested readers are referred to [1].]

5. Deletion

It is possible to delete the root from a k-d tree, although it is rather expensive to do so. In discussing deletion it is sufficient to consider the problem of deleting the root node of a subtree.

If the root, say P, to be deleted has no subtrees then the resulting tree is the empty tree. If P does have descendants, then the root should be replaced with one of those descendants, say Q, that will retain the order imposed by P. That is, all nodes in the *HISON* subtree of P will be in the *HISON* subtree of Q, and likewise for the *LOSON* subtrees. Assume P was a J-discriminator, then Q must be the J-maximum element in the *LOSON* subtree of P (or similarly, the J-minimum element in P's *HISON* subtree). Once Q is found it can serve as the new root, and the only reorganization necessary is to delete Q from its previous position in the tree.

The following recursive algorithm gives a description of one way to accomplish this:

Algorithm *DELETE* (k-d tree deletion)

This recursive algorithm is passed a pointer P to a node in a k-d tree. It deletes the node pointed to by P, and returns a value which is a pointer to the root of the resulting subtree.

D1. [Is P a leaf?] If $HISON(P) = \Lambda$ and $LOSON(P) = \Lambda$ then return Λ; otherwise, set $J \leftarrow DISC(P)$.

D2. [Decide where to get P's successor root.] If $HISON(P) = \Lambda$ then go to D4.

D3. [Get next root from $HISON(P)$.] Set $Q \leftarrow J$-minimum node in $HISON(P)$, $QFATHER \leftarrow$ the father of Q,$QSON \leftarrow$ which son Q is of $QFATHER$ ($QSON$ is either $HISON$ or $LOSON$, s.t. $QSON(QFATHER) = Q$). Go to D5.

D4. [Get next root from $LOSON(P)$.] Set $Q \leftarrow J$-maximum node in $LOSON(P)$, $QFATHER \leftarrow$ the father of Q, $QSON \leftarrow$ which son Q is of $QFATHER$ (see D3).

D5. [Delete Q.] Set $QSON(QFATHER) \leftarrow DELETE(Q)$. (This recursive step will free Q so it can become the new root.)

D6. [Make Q the new root.] Set $DISC(Q) \leftarrow DISC(P)$, $HISON(Q) \leftarrow HISON(P)$, $LOSON(Q) \leftarrow LOSON(P)$. Return Q.

The maximizer/minimizer used in steps D3 and D4 of this algorithm is not described here. It is quite similar to (and uses the same strategy) as the partial match search described in Section 4.1.2 in the case where only one key is specified.

Step D2 as it is presented is a potential source of much trouble. When successive roots of a tree are deleted, each time the successor will be taken from the *HISON* subtree of *ROOT* until the *HISON* subtree is empty, producing a pessimal imbalance. When both the subtrees are nonempty it would probably be better to somehow choose between finding Q in the *LOSON* and *HISON* subtrees, either by "flip-flopping" between them, or by using a pseudorandom number generator to decide. Either of these heuristics should considerably reduce the degeneration inherent in the algorithm as presented.

The algorithm has variable running time in two places: the maximizer/minimizer and the recursive call to itself in step D5. The analysis in Section 4.1.2 tells us that to find the J-extreme node in steps D3 and D4 will use $O(n^{(k-1)/k})$ time. The recursion can become quite expensive (on the order of n levels) as the tree degenerates, but in the average tree the J-nearest node to the root usually does not have very large subtrees. (As in all binary trees, the vast majority of nodes in the tree are leaves, or very close to leaves.) Hence the running time for deletion of the root of a subtree of n nodes will probably be dominated by the maximizer/minimizer, using $O(n^{(k-1)/k})$ time.

We have thus far examined only the worst case—deletion of the root. We can now easily obtain an upper bound for the average cost of deleting a random node from a tree randomly built as described in Section 3. The cost of deleting a node which is the root of a subtree of j nodes is certainly bounded from above by j. Hence to calculate the average cost of deleting a node from tree t, we can merely sum the subtree sizes of t and divide by n. It is easy to show inductively that the sum of subtree sizes of t is $TPL(t) + n$. We showed in Section 3 that the TPL of a randomly built tree is $O(n \log n)$; thus we know that an upper bound for the average cost of deleting a node from a randomly built tree is $O(\log n)$.

6. Optimal Trees

In some circumstances the average behavior described in Section 3 for a k-d tree built by random insertion might not be acceptable. This could be the case if a very large number of searches were going to be made and no nodes were to be inserted or deleted (a static tree), or if the nodes were known to arrive in a destructively nonrandom order. Fortunately it is possible, though costly in running time, to optimize a k-d tree so that all external nodes appear on two adjacent levels.

The following algorithm produces an optimized k-d tree by building a tree such that the number of nodes in the *HISON* subtree of each node differs by at most one from the number of nodes in the *LOSON* subtree. To build an optimized k-d tree, *OPTIMIZE* is called with A as the collection of nodes to comprise the tree and J set to *ROOTDISC*.

Algorithm *OPTIMIZE* (Produce an optimized k-d tree.)

This algorithm is passed a collection of nodes, A, in an appropriate form such as a linked list and a discriminator, J. It returns a pointer to an optimized k-d tree whose root is a J-discriminator.

O1. [Check for null set.] If A is null, return Λ.

O2. [Find median.] Set $P \leftarrow J$-median element of A.

O3. [Split collection.] Set $A_L \leftarrow \{a \in A \mid a$ is J-less than $P\}$, $A_H \leftarrow \{a \in A \mid a$ is J-greater than $P\}$ (A_L is the collection of all points to go into the *LOSON* subtree, A_H is the collection of all points to go in the *HISON* subtree; their cardinalities differ by at most one.)

O4. [Recur.] Set $DISC(P) \leftarrow J$, $M \leftarrow NEXTDISC(J)$, $LOSON(P) \leftarrow OPTIMIZE(A_L,M)$, $HISON(P) \leftarrow OPTIMIZE(A_H,M)$. Return P.

Because of the balancing of number of nodes, an optimized tree has minimum total path length over all k-d trees of n nodes. The maximum path length in an optimized tree of n nodes is $\lfloor \log_2 n \rfloor$. Knuth [4] has shown in his discussion of optimal 1-d trees a result that holds for optimized k-d trees in general: the total path length of an optimized tree of n nodes is

$$TPL_0(n) = \sum_{1 \le i \le n} \lfloor \log_2 i \rfloor = (n+1)q - 2^{q+1} + 2,$$

where $q = \lfloor \log_2 (n+1) \rfloor$.

The maximum path length is also a depth bound for the number of levels of recursion entered through step O4. Let us now examine the amount of time spent on the

jth level of recursion. There are 2^j optimizations going on, each optimizing a subtree of (approximately) $n/2^j$ nodes. Step O2, finding the median of the $n/2^j$ elements, can be accomplished in $O(n/2^j)$ time using the selection algorithm of Blum, Floyd, Pratt, Riverst, and Tarjan [2]. Step O4, the splitting into subcollections, can also be accomplished in $O(n/2^j)$ time, so the total running time on each level of recursion is $2^j \cdot O(n/2^j) = O(n)$. Since the recursion depth is bounded at $\log_2 n$, the total running time of the algorithm is $O(n \log n)$. This is clearly asymptotically optimal, as the algorithm performs the equivalent of a sorting operation.

7. Applications

Let us now consider some situations in which k-d trees might be used. We will study two rather dissimilar specific examples, but we first make two observations about applications in general. First, k-d trees are intended primarily for use in a 1-level store, but using secondary storage (such as disk or drum memory) for overflow might be acceptable if there were few additions and deletions from the file. This factor is becoming less of a restriction as memories rapidly decrease in cost and increase in size. Second, it is necessary for k-d trees to have some minimum number of nodes before they become useful. For example, if the deepest node in a 10-d tree is on the 9th level, one of the keys will never have been used as a discriminator as the tree was being built. A good rule of thumb might be to use k-d trees only if $n > 2^{2k}$. Any application in which there are a multiplicity of keys with no key inherently primary, and which fits the two basic requirements mentioned above, is a potential application for k-d trees. We will investigate some specifics of two such examples.

7.1 Applications in Information Retrieval Systems

Consider a terminal-oriented information retrieval system involving a file whose records are cities on a map, say of the continental United States. The cities could be stored as nodes in a k-d tree with latitude and longitude serving as the keys. Queries could take on many forms. An exact match query might be "What is the city at latitude 43° 3′ N and longitude 88° W?" One could ask the partial match query "What are all cities with latitude 39° 43′ N?" to find all cities on the Mason-Dixon line. To find all cities in the Oklahoma Panhandle one could pose a region query defining a rectangle bounded by latitudes in the range 36° 30′ to 37° and longitudes in the range of 100° to 103°. The nearest neighbor algorithm would be able to answer the query "Which is the closest city to Durham, North Carolina?"

It might be the case that not all cities were to be stored in the file, but only those in which there was some scarce commodity (for instance, an automobile rental agency might wish to ask "What is the closest city to Los Angeles in which there is an available automobile?"). In such a dynamic file, cities could be inserted in the tree as the commodity became available and deleted as it became unavailable; the file could then be optimized if, after much activity, it became too unbalanced. The optimization algorithm could also be used, for example, if the nodes were sorted by state at the time of file creation. Using random insertion in this case would probably produce a terribly unbalanced tree.

7.2 Applications in Speech Recognition

Most speech recognition systems being built now have fairly small vocabularies, on the order of 100 words. As the size of vocabularies increases, k-d trees could play an important role in identifying spoken "unknown utterances" as words in the vocabulary. When an utterance of the speaker enters the system it is decomposed into a fixed number of "features." As an example, the speech might be passed through a bank of bandpass filters, and the amplitude variations as functions of time of each filter's output would together comprise the features. Each word (or "utterance class") in the vocabulary is represented by a "template" which consists of a description of its features. A recognizer must find which template most closely matches the unknown utterance, and report that as the most probable word spoken by the speaker.

If the templates in the vocabulary were stored as records in a k-d tree, with the features serving as the attributes, the nearest neighbor algorithm described in Section 4.2 could be used to efficiently identify a template as being the most likely word spoken. The use of procedures to define the distance measure would permit the choosing of an appropriate similarity measure. The properties of random insertion described in Section 3 indicate that adding to the vocabulary dynamically would involve only a small runtime cost and little degradation in search time. This is important as the system adjusts to a particular user in real time. If much time became available (say, between users), the system could use algorithm *OPTIMIZE* to optimize the tree and thereby guarantee good search times. This suggests that k-d trees could be quite useful for implementing speech recognition systems with large vocabularies.

8. Areas for Further Research

It seems clear the the nearest neighbor algorithm has running time of $O(\log n)$; it would be nice to prove this analytically. The *REGIONSEARCH* algorithm also needs to be analyzed more carefully. Deletion is very costly; perhaps there is a faster way to delete a node. The logarithmic behavior for random insertion when combined with the optimization algorithm will satisfy most users' guaranteed efficiency needs. However, it would be desirable to define some criteria for balancing, such as the AVL criterion for 1-d trees [5].

By definition, the discriminators in k-d trees are strictly alternating (i.e. $NEXTDISC(DISC(P)) = DISC(LOSON(P))$). Having nonstrictly alternating discriminators might enhance the flexibility of k-d trees, and perhaps even lead to criteria for balancing.

9. Conclusions

The k-d tree has been developed as a data structure for the storage of k-dimensional data. The storage cost is two pointers per record in the file. A noteworthy advantage of k-d trees is the fact that a single data structure facilitates many different and seemingly unrelated query types. Random insertion in an n node file is, on the average, an $O(\log n)$ task. Partial match queries with t keys specified can be performed in k-d trees in $O(n^{(k-t)/k})$ time. They are flexible enough to allow any intersection query. Empirical tests show nearest neighbor searches have average running time of $O(\log n)$. Deletion of the root node requires $O(n^{(k-1)/k})$ running time, but deletion of a random node is $O(\log n)$. An optimization algorithm of speed $O(n \log n)$ guarantees logarithmic behavior of the tree. By example, k-d trees were shown to be appropriate data structures for many applications. A good deal of work remains to be done on k-d trees, particularly in the analysis of execution times of some search algorithms.

Acknowledgments. I would like to acknowledge gratefully many hours of fruitful discussion with Peter Deutsch, Ray Finkel, Leo Guibas, Ron Rivest, and Don Stanat. For an exceptionally stimulating environment, much technical support, and ample computation time I am indebted to the Xerox Palo Alto Research Center and the people who work there. Finally, I would like to acknowledge three of my professors—Vint Cerf, Don Knuth, and Ed McCreight. In addition to the technical skill I learned from them, their excitement about computer science and their time invested in me provided motivation to pursue this subject. To these men I owe a great deal.

Received September 1974; revised January 1975

References

1. Friedman, J.H., Bentley, J.L., and Finkel, R.A. An algorithm for finding best matches in logarithmic time. Stanford CS Rep. 75–482.
2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E. Time bounds for selection. Stanford CS Rep. 73–349.
3. Finkel, R.A., and Bentley, J.L. "Quad trees: a data structure for retrieval on composite key." *Acta Informatica 4*, 1(1974), 1–9.
4. Knuth, D.E. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1969.
5. Knuth, D.E. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
6. McCreight, E. Computer Science 144A midterm examination, spring quarter, 1973. Stanford University.
7. Rivest, R.L. Analysis of associative retrieval algorithms. Stanford CS Rep. 74–415.

1975 ACM Student Award
Paper: Second Place

The Digital Simulation of River Plankton Population Dynamics

R. Mark Claudson
Hanford High School
Richland, Washington

This paper deals with the development of a mathematical model for and the digital simulation in Fortran IV of phytoplankton and zooplankton population densities in a river using previously developed rate expressions. In order to study the relationships between the ecological mechanisms involved, the simulation parameters were varied illustrating the response of the ecosystem to different conditions, including those corresponding to certain types of chemical and thermal pollution. As an investigation of the accuracy of the simulation methods, a simulation of the actual population dynamics of Asterionella in the Columbia River was made based on approximations of conditions in that river. Although not totally accurate, the simulation was found to predict the general annual pattern of plankton growth fairly well and, specifically, revealed the importance of the annual velocity cycle in determining such patterns. In addition, the study demonstrates the usefulness of digital simulations in the examinations of certain aquatic ecosystems, as well as in environmental planning involving such examinations.

Key Words and Phrases: digital simulation, mathematical modeling, plankton population dynamics, phytoplankton, zooplankton, river ecosystems, ecological mechanisms, environmental simulation, modeling ecosystems, pollution, environmental impact, environmental planning

CR Categories: 3.12, 3.19

This paper was awarded Second Place in ACM's 1975 George E. Forsythe Student Paper Competition.

Author's present (home) address: 402 Sierra Street, Richland, WA 99352.