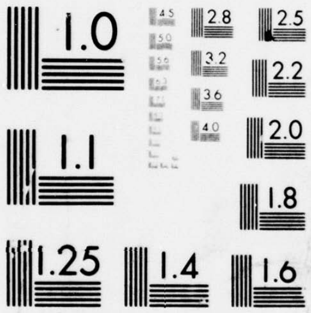
AD-A061 626 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
MULTIDIMENSIONAL BINARY SEARCH TREES IN DATABASE APPLICATIONS.(U)
SEP 78 J L BENTLEY N00014-76-C-0370
UNCLASSIFIED CMU-CS-78-139 NL

1 OF 1
AD
A061626

END
DATE
FILMED
2 - 79
DDC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

CMU-CS-78-139

B.S. 12

MULTIDIMENSIONAL BINARY SEARCH TREES IN DATABASE APPLICATIONS

Jon Louis Bentley
Departments of Computer Science and Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

8 September 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

(14) CMU-CS-78-139

(6)

MULTIDIMENSIONAL BINARY SEARCH TREES IN DATABASE APPLICATIONS

(10) Jon Louis Bentley
Departments of Computer Science ~~and Mathematics~~
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

(11) 8 Sept[redacted]78 (12) 18p.

(9) Interim rept.

(15) N00014-76-C-0370

Abstract

The multidimensional binary search tree (abbreviated k-d tree) is a data structure for storing multi-key records. This structure has been used to solve a number of problems in geometric data bases arising in statistics and data analysis. The purposes of this paper are to cast k-d trees in a database framework, to collect the results on k-d trees which have appeared since the structure was introduced, and to show how the basic data structure can be modified to facilitate implementation in large (and very large) databases.

The research in this paper was supported in part by the office of Naval Research under contract N00014-76-C-0370.

Table of Contents

1 Introduction

It is no secret that the designer of a database system faces many difficult problems and is armed with only a few tools for solving them. Among those problems are reliability, protection, integrity, implementation, and choice of query languages. In this paper we will examine a solution to yet another problem which the database designer must face (while keeping the above problems in mind): the design of a database system which facilitates rapid search time in response to a number of different kinds of queries. We will confine our attention to databases of "fixed length records without pointers"; specifically we assume that we must organize a *file* F of N *records*, each of which contains k *keys*. Much previous research has been done on problems cast in this framework; the interested reader is referred to Lin, Lee and Du [1976], Rivest [1976], and Wiederhold [1977, Chapters 3-4] for discussions of many different approaches.

In this paper we will examine a particular data structure, the ***multidimensional binary search tree***, for its suitability as a tool in database implementation. The multidimensional binary search tree (abbreviated *k-d tree* when the records contain k keys) was introduced by Bentley [1975]. The k-d tree is a natural generalization of the well-known binary search tree to handle the case of a single record having multiple keys. It is a particularly interesting structure from the viewpoint of database design because it is easy to implement and allows a number of different kinds of queries to be answered quite efficiently. The original exposition of k-d trees in Bentley [1975] was cast in geometric terms, and since that time the k-d tree has been used to solve a number of problems in "geometric" data bases arising in data analysis and statistics. The purposes of this paper are to cast k-d trees in a database framework, to collect the results on k-d trees which have appeared since the structure was introduced, and to show how the basic data structure can be modified to facilitate implementation in large (and very large) databases.

Since k-d trees are a natural generalization of the standard binary search trees we will review that well-known data structure in Section 2. In Section 3 we develop the k-dimensional binary search tree (k-d tree). We describe how different types of searches can be performed in Section 4 and discuss the maintenance of k-d trees in Section 5. Section 6 faces the problems of implementing k-d trees on different storage media, and directions for further work and conclusions are offered in Sections 7 and 8.

2 One-dimensional Binary Search Trees

In this section we will briefly review binary search trees; a more thorough exposition of this data structure can be found in Knuth [1973, Section 6.2]. Figure 2.1.a is an illustration of a binary search tree representing the numerically-valued keys 31, 41, 15, and 92 (which were inserted in that order). In Figure 2.1.b the additional key 28 has been inserted. The defining property of a binary search tree is that for any node x the key values in the left subtree of x are all less than the key value of x and likewise the key values in the right son are greater than x's. To search to see if a particular value y is currently stored in a tree one starts at the root and compares y to the value of the key stored at the root, which we can call z. If y equals z then we have found it, if y is less than z then our search continues in the left son, and if y is greater than z then we continue in the right son. To insert an element we apply the searching process until it "falls out" of the tree and then change the last "null pointer" observed to point to the new element.

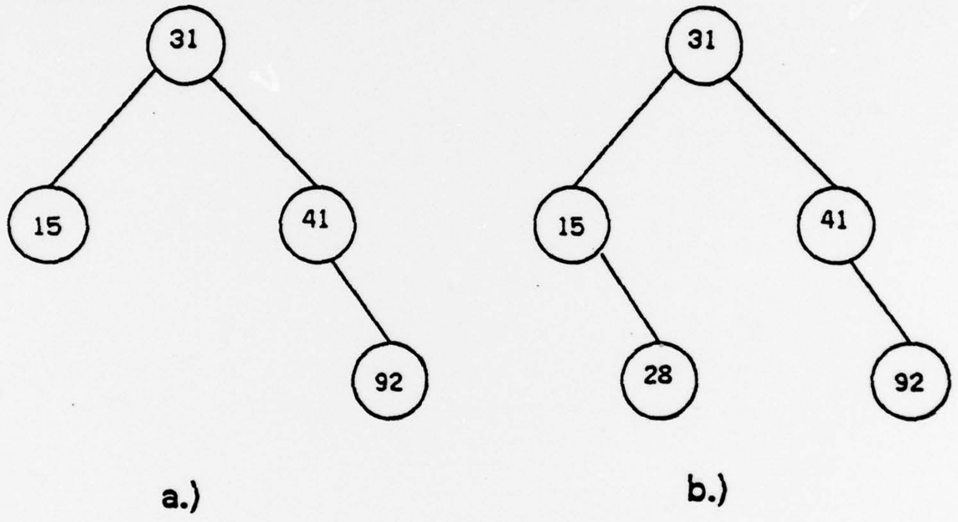
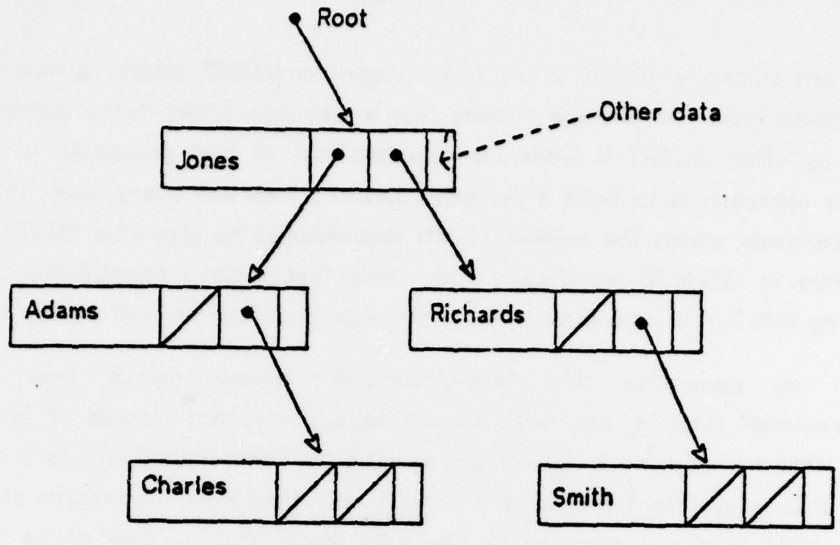
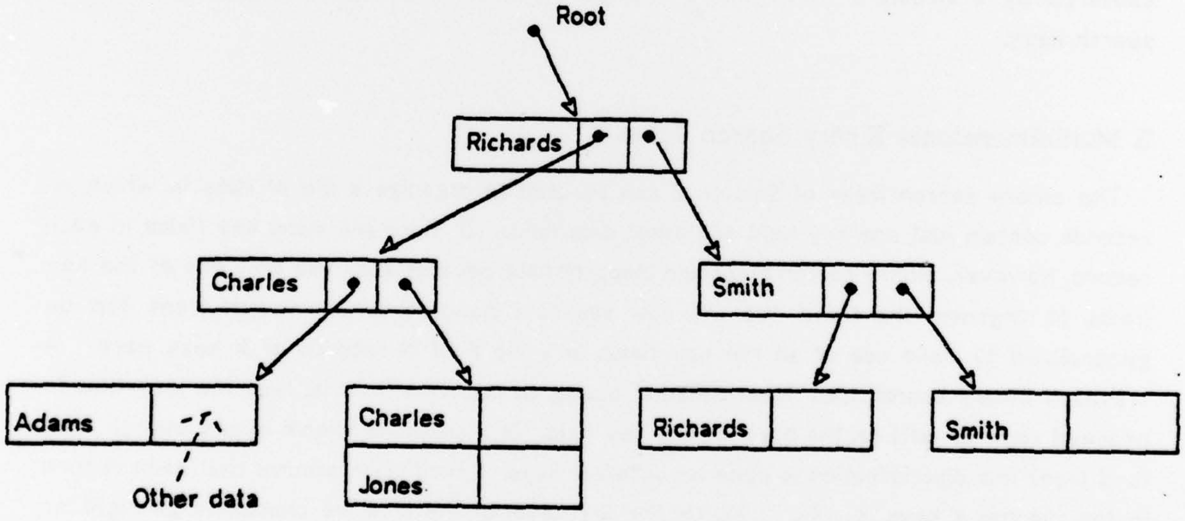


Figure 2.1. Two abstract binary search trees.

The abstract binary search tree can be implemented on a computer in many different ways. The most popular representation of a node in a tree is what we will call the *homogeneous*. In this representation a node consists of a KEY field (which holds the single key defining the record), LEFT and RIGHT son pointers, and additional fields which hold the rest of the data associated with the record. Note that in this approach a node in the tree serves two distinct purposes: representation of a record and direction of a search. These two functions are separated in a *nonhomogeneous* binary search tree, in which there are two kinds of nodes: internal and external. An internal node contains a KEY field and LEFT and RIGHT son pointers, but no data; all records are held in external nodes which represent sets of records (or perhaps individual records). In nonhomogeneous trees it is important to make the convention that if a search key is equal to the value of an internal node, then the search continues in the right subtree. Homgeneous trees are superior to nonhomgeneous trees when the elements of the tree are inserted succesively. If the elements are to be built once-for-all into a perfectly balanced tree then the nonhomogeneous tree is usually preferred. Figure 2.2 depicts a set of records stored in the two kinds of trees. Another situation in which the nonhomgenous tree is superior to the homgeneous tree is when the records are to be stored on a secondary storage device. In that case the nonhomgeneous tree offers the advantage that entire records do not have to be read into main memory to make a branching decision when only the key is required; we will cover this point in detail in Section 6.



a.) Homogeneous



b.) Nonhomogeneous

Figure 2.2. Binary search tree implementations.

In the above discussion we have alluded to a number of algorithms for performing operations on binary search trees. Algorithm SEARCH tells if a record containing a given key is stored in the tree; Knuth has shown that this algorithm takes O(lg N) expected time if N elements are currently stored in the tree. Algorithm INSERT inserts a new node into a (homogeneous) tree. Its average running time is also logarithmic; if one builds a tree of N elements by using INSERT N times the expected cost of that procedure is O(N lg N). An alternative approach is to build a perfectly balanced tree (for every node, the number of right descendants equals the number of left descendants) by algorithm BUILD; this can be accomplished in O(N lg N) worst-case time. Note that while a homogenous tree may be balanced by BUILD, it is imperative that a nonhomogeneous tree be built by this algorithm.

Before we generalize this "one-dimensional" binary search tree to become "multidimensional" (that is, deal with several keys per record instead of just one) it is important that we stop for a moment and examine the "philosophy" of binary search trees. These structures perform three tasks at once. Firstly, they *store* the records of a given set. Secondly, they impose a *partition* on the data space (that is, they divide the line into segments). Thirdly, binary search trees provide a *directory* which allows us to locate rapidly the position of a new point in the partition by making a logarithmic number of comparisons. In the next section we will see how these essential features of binary search trees can be captured by a structure which allows a single record to be retrieved by many different search keys.

3 Multidimensional Binary Search Trees

The binary search trees of Section 2 can be used to organize a file of data in which all records contain just one key field and other data fields. If there are many key fields in each record, however, binary search trees are inappropriate because they use only one of the key fields to organize the tree.[1] We will now see how standard binary search trees can be generalized to make use of all the key fields in a file F of N records of k keys each. A standard binary search tree "discriminates" during an insertion (that is, tells the insertion to proceed right or left) on the basis of one key field. In a multidimensional binary search tree (k-d tree) this discrimination is done on different keys. Specifically, assume that each record in the file has k keys, K_1, K_2, ..., K_k. On the first level of the tree we choose to go right or left when inserting a new record by comparing the first key (K_1) of the new record with the first key of the record stored at the root of the k-d tree (assume a homogenous representation). At the second level of the tree we use the second key as the discriminator, and so on to the k-th level. Finally at the (k+1)-st level of the tree we "wrap around" and use the first key as the discriminator again. We illustrate this concept in Figure 3.1. Records in that tree each contain two keys: name (K_1) and age (K_2). Note how every record in the

[1]We say that a field is a key field if a query can refer to it. For example, a set of records might each contain employee number, department number, and salary fields; if queries can refer only to department number and salary then those two fields are keys while employee number is data.

left subtree of the root has a name field less than the root's, and likewise every record in the right subtree has a greater name field. On the second level right subtrees have greater age values.

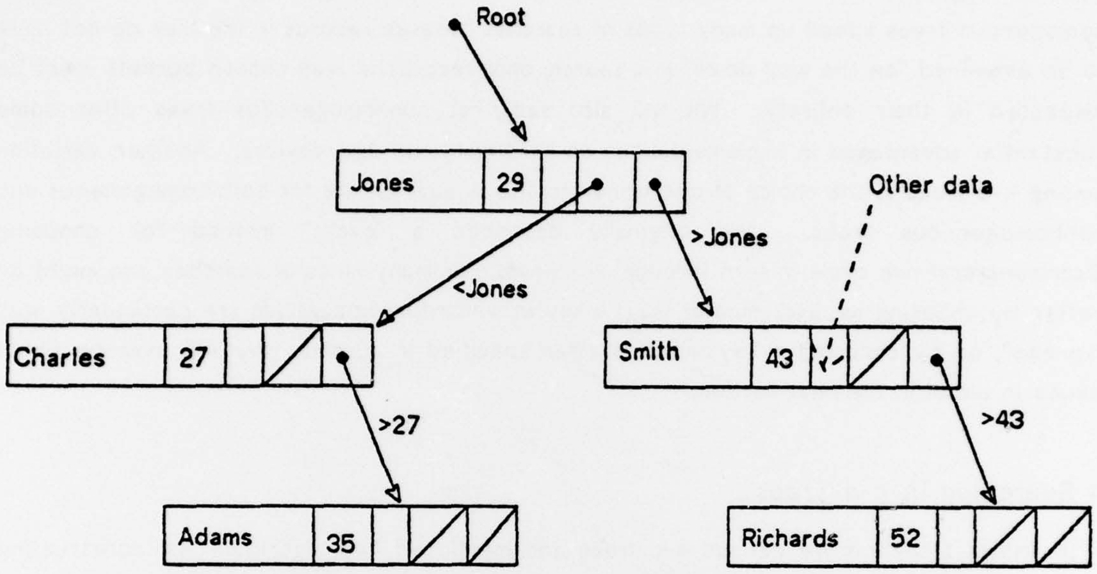


Figure 3.1. A homogeneous 2-d tree: K_1 is name and K_2 is age.

We can now give a more formal definition of k-d trees. A homogeneous k-d tree is a binary tree in which each record contains k keys, some data fields (possibly), right and left son pointers, and a discriminator which is an integer between 1 and k, inclusive. In the most straightforward version of k-d trees all nodes on level i have the same discriminator, namely (i mod k). The defining property of k-d trees is that for any node x which is a j-discriminator, all nodes in the left subtree of x have K_j values less than x's K_j value, and likewise all nodes in the right subtree have greater K_j value. To insert a new record into a k-d tree we start at the root and search down the tree for its position by comparing at each node visited one of the new record's keys with one of the keys of that node, namely the one specified by the discriminator. Bentley [1975] has shown that if a set of N "random" records are inserted into a k-d tree then it will require approximately 1.386 lg N comparisons to insert the N-th record, on the average; the expected cost of performing all N insertions is O(N lg N).

As there are many implementations of one-dimensional binary search trees, so there are many implementations of k-d trees. The k-d trees which we have described above correspond to the homogenous binary search trees; it is also possible to define nonhomogeneous k-d trees. Internal nodes in such k-d trees contain only a discriminator (an integer between 1 and k), one key value (chosen by the discriminator), and left and right son pointers. All records in nonhomogeneous k-d trees are stored in external nodes or "buckets".

(This version of k-d trees was originally proposed by Friedman, Bentley and Finkel [1977].) A file F of N records can be built into a perfectly balanced homogenous k-d tree in O(kN lg N) time; an algorithm for accomplishing this is given in Bentley [1975]. We will see that homogenous trees speed up many types of searches because records in the tree do not have to be examined "on the way down" in a search; only records in well chosen buckets must be inspected in their entirety. We will also see that nonhomogeneous trees offer some substantial advantages in implementations on secondary storage devices. Another variation among k-d trees is the choice of discriminators; this is appropriate for both homogoneous and nonhomogeneous trees. We originally described a "cyclic" method for choosing discriminators--we cycle in turn through all k keys. For many kinds of searches one might do better by choosing as discriminator (say) a key in which the data values are particularly well "spread", or by choosing a key which is often specified in queries. We will examine these issues in detail in the next section.

4 Searching in k-d Trees

In the last section we defined k-d trees and mentioned two algorithms for constructing them: by repeated insertion and a "once-for-all" algorithm that produces a perfectly balanced tree. In this section we will examine a number of different algorithms for searching k-d trees, each appropriate in answering a certain kind of query. We will discuss four particular types of searches in detail, and then briefly mention other types of searching possible in k-d trees.

4.1 Exact Match Queries

The simplest type of query in a file of k-key records is the exact match query: is a specific record (defined by the k keys) in the file? To answer this query we proceed down the tree, going right or left by comparing the desired record's key to the discriminator in the node, just as in the insertion algorithm. In the homogeneous version of k-d trees we will either find the record in a node on the way down or "fall out" of the tree if the record is not present. In the nonhomogeneous version we will be directed to a bucket and can then examine the records in that bucket to see if any are the desired. The number of comparisons to accomplish an exact match search is O(lg N) in the worst case if the tree is perfectly balanced; it is also O(lg N) on the average for randomly built trees.

4.2 Partial Match Queries

A more complicated type of query in a multikey file is a "partial match query with t keys specified". An example of such a query might occur in a personnel file: report all employees with Length-of-Service = 5 and Classification = Manager, ignoring all other keys in the records. In general we specify values for t of the k keys and ask for all records which have those t values, independent of the other k-t values. Bentley [1975] describes an algorithm for searching a k-d tree to answer such queries, which we will now sketch. We start the search by visiting the root of the k-d tree. Whenever we visit a node of the k-d tree which discriminates by j-value we check to see if the value of the j-th key is specified in the query; if it is, then we need only visit one of the node's sons (which son is determined by comparing the desired K_j with that node's K_j value). If K_j is not one of the t keys specified, then we must recursively search both sons. Bentley [1975] shows that if t of k keys are specified then the time to do a partial match search in a file of N records is $O(tN^{1 - t/k})$.[1] As an example, if 4 of 6 keys are specified in a partial match search of one million records, then only approximately 400 records will be examined during the partial match search.

4.3 Range Queries

In a range query we specify a range of values for each of the k keys, and all records which have every value in the proper ranges are then reported as the answer. For example, we might be interested in querying a student data base to find all students with Grade Point Average between 3.0 and 3.5, Parent's Income between $12,000 and $20,000, and Age between 19 and 21. This problem arises in many applications; Bentley and Friedman [1978] mention some of those applications and survey the different data structures currently used for solving the problem.

It is easy to answer a range query in a k-d tree; the algorithm to do so is similar to the partial-match searching algorithm. As we visit a node which is a j-discriminator we compare the j-value of that node to the j-range of the query. If the range is entirely below the value then the search continues on the left son, if it is entirely above then the search visits the right son, otherwise both sons are recursively searched. Lee and Wong [1977] have analyzed the worst-case performance of that algorithm and have established that the time required to perform a range search is never more than $O(N^{1 - 1/k} + F)$, where F is the number of points found in the range. Although it is nice to know that things can never get *really* bad, the average case of searching is much better. Bentley and Stanat [1975] reported results for a data structure very similar to k-d trees which imply that the expected time for range searching in k-d trees is $O(\lg N + F)$. It is difficult to analyze the exact performance of range searching because it is so dependent on the "shape" of the particular query, but empirical evidence strongly suggests that k-d trees are very efficient.

[1]For $t < k$. If $t = k$ then this is an exact match search, and $O(\lg N)$ time is required.

4.4 Best Match Queries

In some database applications we would like to query the database and find that it contains exactly what we are looking for; a builder might hope to find that he has in his warehouse exactly the kind of steel beams he needs for the current project. But often the database will not contain the exact item, and the user will have to settle for a similar item. The most similar item to the desired is usually called the "best match" or the "nearest neighbor" to the desired. In information retrieval systems we hope for a book that discusses all ten topics in our list, but we must settle for one (say) that mentions only eight. Friedman, Bentley and Finkel [1977] showed how k-d trees can be used to answer such best match queries (where "best" can be defined by many different kinds of "distance functions"). Their algorithm depends on choosing the discriminators in a sophisticated fashion. They showed that the expected amount of work to find the M best matches to a given record is proportional to lg N + M in any fixed dimension. Their algorithm was implemented in FORTRAN for applications in geometric data bases and empirical tests showed that their algorithm is orders of magnitude faster than the previous algorithms, for practical problem sizes. Since that time Zolnowsky [1978] has analyzed the worst case of nearest neighbor searching in k-d trees and has shown that although any particular search can be rather expensive, if a search for the nearest neighbor of every point in some fixed set is performed then the worst-case cost of searching will average to $O((\lg N)^k)$.

4.5 Other Queries

The four types of queries we have already investigated are the most common queries in fixed-format database applications. Other query types do arise, however, and k-d trees can often be used to answer them. Bentley [1975] gives a procedure which allows k-d trees to answer "intersection" queries which call for all records satisfying properties which can be tested on a record-by-record basis. The best match algorithm of Section 4.4 finds the best match to a particular record; that can be modified to find the best match to a more general description (such as a range, for example). An interesting modification to the basic idea of k-d trees was made by Eastman [1977], who developed a binary tree data structure appropriate for nearest neighbor searching in document retrieval systems.

5 Maintaining k-d Trees

In Section 3 we defined the k-d tree and in Section 4 we described different algorithms for searching k-d trees; in this section we will investigate the problems of maintaining k-d trees. Specifically we will discuss the problems of *building* a set of records into a k-d tree, *inserting* a new element into an existing tree, and *deleting* an existing element from a tree. We will discuss these problems in the two cases of homogeneous and nonhomogeneous trees.

We have already seen the insertion algorithm for homogeneous trees in Section 3. We mentioned that a perfectly balanced tree can be built in O(kN lg N) time; an algorithm to do so is given by Bentley [1975], which we will now sketch for the case of cyclic homogeneous trees. The first step of the algorithm finds the median K_1 value of the entire set (that element greater than one half of the K_1 values and less than the other half). We then let the record corresponding to that element be the root of the entire tree and put the N/2 elements with lesser K_1 value in the left subtree and the other N/2 elements in the right subtree. At the next level we find for each of those two subfiles of N/2 points their K_2 medians, and use those two records as the roots of the two subtrees. This process continues, finding the medians at each level and partitioning around them. If a fast algorithm is used to find medians, then this can be accomplished in O(kN lg N) time. Deletion of a node in a homogeneous k-d tree seems to be a fairly difficult problem. Bentley [1975] gives an algorithm which can delete a node in $O(N^{1 - 1/k})$ worst-case time. Fortunately, the average running time of that deletion algorithm is much less: O(lg N).

The problems of maintaining a nonhomogeneous k-d tree (compared to a homogeneous) seem to be much easier on the average but more difficult when considering the worst case. Recall that there are two types of nodes in a nonhomogeneous k-d tree: internal nodes which contain only discriminators and pointers, and external nodes (or buckets) which contain sets of records. Friedman, Bentley and Finkel [1975] report that in best match searching the optimal number of records per buckets is about a dozen; this will probably be a reasonable number for many applications. The algorithm that we sketched above for building a homogeneous tree can be applied almost immediately to build a nonhomogeneous tree; its worst-case running time is also O(kN lg N). A good average-case strategy for insertion and deletion in a nonhomogeneous k-d tree is merely to insert the record into or delete the record from the bucket in which it resides; both of these operations can be accomplished in logarithmic time. If the insertions and deletions are scattered almost equally throughout the file then this method will produce very good behavior. If the resulting tree ever becomes too unbalanced for a particular application, then the optimization algorithm could be run again to produce a new optimal tree. (This is especially appealing if there are periods of inactivity in the database, such as at night in a banking system.) Another benefit of nonhomogeneous trees is that if "multiple writers" are used to perform insertions and deletions then they will have to "lock" only the bucket containing the current record (and no nodes higher in the tree).

6 Implementing k-d Trees

Our discussion of k-d trees so far in this paper has assumed that the cost of going from a node to its son is constant for all nodes, but this is not true in all implementations of k-d trees. In this section we will investigate the problems of implementing k-d trees on various storage devices. If k-d trees are to be implemented in the main memory of a computer then either the homogeneous or nonhomogeneous versions will serve well, with links between sons implemeted as pointers to records.

If k-d trees are to implemented on a secondary storage device such as disk then one would probably use nonhomogeneous trees.[1] As an example, assume that we had a file F of ten million records, each of five keys. If we allocate ten records in each bucket then there will be one million buckets in the system; this implies that the height of the "internal" part of the tree is twenty, since $\log_2 1{,}000{,}000 = 20$. Because there are too many internal nodes in the tree to store in the main memory, we must store both internal and external nodes on the disk. We will accomplish this by grouping together on the same disk pages internal nodes which are "close" in the tree (see Knuth [1973, Section 6.2.4] for the application of this technique to one-dimensional trees). This process is illustrated in Figure 6.1. If the discriminators are of reasonable length then compression techniques can be used to store an internal node in (say) ten bytes of storage; this implies that we can store one thousand internal nodes (or ten levels of the tree) on a ten-thousand-byte disk page. Thus there is a distance of only two pages between the root of the tree and any external node, so if the page containing the root is kept in main memory at all times, any record can be accessed in only two page transfers from disk.

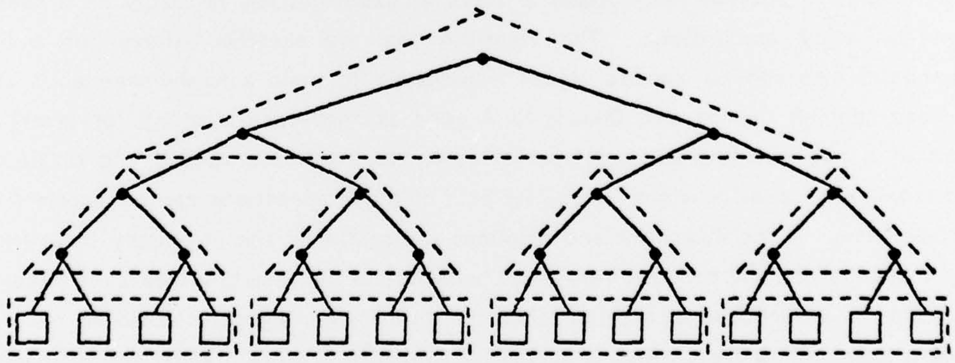


Figure 6.1. Disk pages denoted by dashed lines.

There are a few minor observations which can significantly improve the performance of k-d trees implemented on secondary storage devices. We saw above how it is crucial for the

[1]For a description of an implementation of homogeneous trees on disk storage see Williams *et al* [1975].

internal nodes to require as little space as possible. One means of achieving this space reduction is through key compression. Instead of storing the entire discriminating key in a node, we need only store enough of its first bits to allow us to later test whether to go right or left. For example, if the discriminator in a name field is "Jefferson" it might be sufficient to store only "Jeffe". Likewise, lower in the tree it is probably not necessary to store some of the leading bits of a discriminator. Another device which can be used to save space on the pages holding internal nodes is the "implicit" binary tree scheme which obviates the need for pointers in defining a binary tree (this is often called a heap). The root of the tree is stored in position 1 of an array and the left and right sons of node i are found in locations 2i and 2i + 1, respectively. It might be that finding the exact median in building a k-d tree is very expensive. If this is so, then an approximation to the exact median would probably serve just as well as the the exact discriminator. Weide [1978] has given an algorithm which finds approximate medians of large data sets very efficiently; his algorithm should probably be used in such an application. During a search in a k-d tree implemented on secondary storage only a relatively few pages will be kept in main memory at a time; a "least recently used" page replacement algorithm should probably be used to decide which old page to release when reading in a new page.

The above scheme appears very promising for many different applications of k-d trees on secondary storage devices. Though we analyzed the scheme only for exact match searching, it should also work very well for all of the other search algorithms described in Section 4. Note the important role that nonhomogeneous k-d trees play in this secondary storage scheme: because the internal nodes are very small compared to the size of the entire records, many of them can reside on one disk page, drastically reducing the required number of disk accesses.

7 Further Work

Although much research has been done on k-d trees since they were introduced in 1975, there are still many further areas which need work. On the practical side it is important that k-d trees be implemented in real database systems to see how the theory relates to practice. Another fascinating problem that needs investigation is methods for choosing discriminator values. Naive k-d trees chose discriminators cyclically, and the k-d trees of Friedman, Bentley and Finkel [1977] chose as the discriminating key that key with the largest spread in its subspace of the key space. For many database applications, however, it is important to choose as discriminator a key which is used often in queries. Some heuristics proposed by Bentley and Burkhard [1976] for "partial match tries" might be useful in the context of k-d trees.

Perhaps the most outstanding open theoretical problem on k-d trees is the question of whether they can be "balanced". For one-dimensional binary search trees there are a number of balancing schemes (such as the AVL trees described by Knuth [1973]) which allow insertions and deletions in logartihmic worst-case time while ensuring that the tree never becomes unbalanced. It is not known whether or not there exist appropriate "balancing acts" for k-d trees.

8 Conclusions

In this paper we have investigated multidimensional binary search trees from the viewpoint of the database designer. The structure was defined in Section 3, and in Section 4 we saw that it supports a number of different kinds of queries. This is an especially important feature for data base applications; it is essential that different query types be handled and it is most unattractive to have to store different data structures representing the same file. In Section 5 we saw a number of different maintenance algorithms for k-d trees. The maintenance algorithms for nonhomogeneous k-d trees are particularly simple to code and are very efficient on the average. In Section 6 we investigated the implementation of k-d trees on secondary storage devices and saw that they can indeed be implemented very efficiently. This implies that k-d trees can be used effectively in large and very large databases. Some areas for further research were described in Section 7.

This paper represents the one of the first attempts to apply k-d trees to the problems which database designers must face. Although we have only scratched the surface of the application of this data structure in this problem domain, it appears that multidimensional binary search trees will be an important addition to the practicing database designer's tool bag.

References

Bentley, J. L. [1975]. "Multidimensional binary search trees used for associative searching," *Communications of the ACM 18*, 9, September 1975, pp. 509-517.

Bentley, J. L. and W. A. Burkhard [1976]. "Heuristics for partial match retrieval data base design," *Information Processing Letters 4*, 5, February 1976, pp. 132-135.

Bentley, J. L. and J. H. Friedman [1978]. "Algorithms and data structures for range queries, " *Proceedings of the Computer Science and Statistics: Eleventh Annual Symposium on the Interface*, March 1978, pp. 297-307.

Bentley, J. L. and D. F. Stanat [1975]. "Analysis of range searches in quad trees," *Information Processing Letters 3*, 6, July 1975, pp. 170-173.

Eastman, C. M. [1977]. A tree algorithm for nearest neighbor search in document retrieval systems. Unpublished Ph.D. thesis, University of North Carolina, Chapel Hill, North Carolina. 111 pp.

Friedman, J. H., J. L. Bentley, and R. A. Finkel [1977]. "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software 3*, 3, September 1977, pp. 209-226.

Knuth, D. E. [1973]. *The art of computer programming, volume 3: Sorting and searching*, Addison-Wesley, Reading, Mass.

Lee, D. T. and C. K. Wong [1977]. "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica 9*, 1, pp. 23-29.

Lin, W. C., R. C. T. Lee, and H. C. Du [1976]. Towards a unifying theory for multi-key file systems. Report, National Tsing-Hua University, Taiwan, Republic of China, 67 pp.

Rivest, R. L. [1976]. "Partial match retrieval algorithms," *SIAM Journal on Computing 5*, 1, March 1976, pp. 19-50.

Weide, B. W. [1978]. "Space-efficient on-line selection algorithms," *Proceedings of the Computer Science and Statistics: Eleventh Annual Symposium on the Interface*, March 1978, pp. 308-311.

Wiederhold, G. [1977]. *Database design*, McGraw-Hill, New York.

Williams, E. H. Jr., A. Vaughn, B. McLaughlin, and M. Buchanan [1975]. PABST program logic manual. Unpublished class project, University of North Carolina, Chapel Hill, North Carolina.

Zolnowsky, J. E. [1978]. Topics in computational geometry. Unpublished Ph.D. thesis, Stanford University, Stanford, California. 53 pp.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-139	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MULTIDIMENSIONAL BINARY SEARCH TREES IN DATABASE APPLICATIONS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Louis Bentley		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Schenley Park, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE Sept. 8, 1978
		13. NUMBER OF PAGES 19
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Rellease; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The multidimensional binary search tree (abbreviated k-d tree) is a data structur for storing multi-key records. This structure has been used to solve a number of problems in geometric data bases arising in statistics and data analysis. The purposes of this paper are to cast k-d trees in a database framework, to collect the results on k-d trees which have appeared since the structure was introduced, and to show how the basic data structure can be modified to facilitate implementation in large (and very large) databases.

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)